Runtime Aggregation of Recursion Relations

Joel Saltz and Harry Berryman

YALEU/DCS/TR-677
January 1989

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

4

Yale University
Department of Computer Science

Runtime Aggregation of Recursion Relations

Joel Saltz and Harry Berryman

YALEU/DCS/TR-677
January 1989

This work has been supported in part by the U.S. Office of Naval Research under Grant N00014-86-K-0310

Runtime Aggregation of Recursion Relations [*]

Joel Saltz
and *Harry Berryman*
Department of Computer Science
Yale University
New Haven, CT 06520

January 25, 1989

Abstract

In many modern algorithms, relatively regular problems are encoded using flexible general purpose data structures. To obtain satisfactory performance on distributed memory architectures, it is often necessary to reconstruct and exploit the underlying dependency structure. We present a method to partition loops that have runtime dependencies that resemble uniform recurrence equations. Loops of this type are often found, among other places, in solving sparse triangular linear systems used for preconditioning in Krylov space iterative linear system solvers.

1 Introduction

The focus of this paper is to examine a method of partitioning loop structures arising from problems such as sparse system solvers whose irregularity can result in poor performance on distributed memory machines. In these problems, crucial details concerning loop dependences are encoded in data structures, rather than being explicitly represented in the program.

Many modern algorithms have inner loops that specify recurrence equations whose dependency patterns are not determined until program execution, i.e. the dependencies in the recurrence equations are *implicitly specified.* An extremely important example of implicitly specified recurrence equations are encountered when solving sparse triangular systems arising from incomplete factorizations of matrices formed from meshes of partial differential equations. Krylov space iterative linear system solvers preconditioned using incompletely factored matrices are frequently used to solve partial differential equations. Because these algorithms are iterative in nature, one must repeatedly solve sparse triangular systems. Some operator splitting methods used in solving time dependent partial differential equations also require triangular system solutions at every timestep. [3]. In both iterative solvers and operator splitting methods, the recurrence equations arise from meshes of partial differential equations and consequently have patterns of dependencies that reflect the structure of the underlying mesh.

There are a variety of other algorithms such as sparse incomplete numeric factorization and dynamic programming that can also be written as recurrence equations with implicitly specified dependencies.

*This work was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0310

The partitioning algorithm is presented in the context of several running example dependency graphs. In section 2 we describe how such a graph would be partitioned if global information concerning the dependency structure were known. In section 3 we describe both how our partitioning algorithm operates and the relationship between our algorithm and the partitioning method described in section 2. This partitioning algorithm has been implemented on the Intel iPSC/1 and iPSC/2, references are provided that outline experimental results we have obtained using this method.

2 Motivation for Partitioning Algorithm

To motivate the algorithm that will be presented, we first consider how we might partition a two-dimensional recurrence equation

$$y_{i,j} = ay_{i-1,j} + by_{i,j-1} + cy_{i+1,j-1} \tag{1}$$

In Figure 2 we depict the dependencies characterizing the above recursion equation defined on a 12 by 4 point rectangle in the upper quadrant. The dependence pattern in a uniform recurrence equation can be described by *direction vectors* that represent the i, j coordinate difference between the left hand side of the equation and the terms on the right hand side. In equation 1, those direction vectors are $w_1 = (1, 0)$, $w_2 = (0, 1)$ and $w_3 = (-1, 1)$.

This system has 16 computational wavefronts; variables satisfying $i + 2j = k$ for constant values of k can be solved for concurrently. If we partition the work in each wavefront equally between two processors we will have 15 communication startups.

Aggregating or clustering the recursion equations to be solved can lead to a substantial reduction in the costs of communication. Equations can be grouped into sets or *clusters*, each cluster is treated as a single schedulable unit. Information is available from a cluster only when all of the work assigned to the cluster has been completed.

The computation can be carried out by computing consecutive lines of domain points parallel to $w_1 = (1, 0)$ or a sequence of lines parallel to $w_3 = (-1, 1)$. In Figure 2 we see that the lines parallel to w_1 run parallel to the x axis, the lines parallel to w_3 have a diagonal orientation. We can compute values for all points i, j in the subdomain satisfying $j \leq k_1$ or where $i + j \leq k_2$ for positive constants k_1 or k_2. These lines are called *separating hyperplanes* [6].

Sets of consecutive lines or *tubes* can be used in forming clusters. In figure 3, we slice the domain presented in figure 2 into consecutive tubes of two adjacent lines parallel to w_1. Within each tube, we form clusters by taking two adjacent lines parallel to w_2. We now have only 8 computational phases requiring at most 7 communication startups.

In many numerical programs, data structures used are very flexible. Crucial details concerning loop dependences are encoded in these structures, rather than being explicitly represented in the program. This method of programming proves to be very advantageous because it facilitates: (1) the production of portable general subroutines and (2) the judicious application of computational resources to irregular numerical problems. Thus, in many cases of practical interest, we are presented with nest of loops representing a recursion equation with some form of adjacency list. While the topic of parallelizing and partitioning systems of uniform recurrence equations has been well studied, see for instance [7], [5] this work all assumes that global knowledge of the dependency structure of a problem.

```
do i=1,n
   do j=low(i),high(i)
     x(i) = x(i) + s(j)*x(pointer(j))
   end do
end do
```

Figure 1: Sparse Form of Recursion Equations

An example of this kind of flexible data structure is given by figure 1 this figure depicts a Fortran code segment that could carry out the same computations as specified in equation 1 (subject to some appropriate set of boundary conditions).

To solve equation 1 using the program in Figure 1 on an n by m domain subject to some set of boundary conditions, we could represent $y_{i,j}$ by $x(i+nj)$. The elements of **x** required to compute $x(i)$ are specified by locations `low(i)` and `high(i)` in array **`pointer`**. For an index i representing a domain point away from the domain boundary, the required elements of x are $x(i-n)$,$x(i-n+1)$ and $x(i-1)$. The array **s** in Figure 1 stores in the appropriate order, copies of the multiplicands a, b and c. Recursion equations whose dependencies are determined by adjacency list type data structures will be termed *implicitly specified* recursion equations (ISRE).

There is a key difference between clustering recurrence relations, and partitioning various possibly irregular time-dependent problems such as solving partial differential equations on irregular meshes, time driven simulations (e.g. battlefield simulations) [4],citeberger87. Time-dependent problems involve a fixed sequence of computational steps. Each step is composed of parallelizable work, the volume of which is independent of the partitioning. The communication flux between steps is determined by the workload partitioning. When we partition recurrence equations we *determine* the collection of work that will be carried out during each computational step, and the number of steps. In many ways this is a more difficult problem, because our clustering simulataneously affects the degree of available parallelism, and the communication overheads.

The clustering method is critical. Consider the explicitly defined recursion equation

$$z_{i,j} = a_{i,j}z_{i,j-1} + b_{i,j}z_{i-1,j}, \qquad (2)$$

on an $X \times Y$ point square (subject to suitable boundary conditions). It is commonly known that one can concurrently solve for variables along anti-diagonals, i.e. variables $y_{i,j}$ satisfying $i+j=k$ for positive k [8]. The anti-diagonals constitute *phases* of the computation. Each phase suffers communication startup costs, to propagate solutions needed to execute the subsequent phase. If we consider each point in a phase to be a schedulable unit of work, this method employs no clustering whatsoever. By contrast, we can cluster point solutions in a way that preserves parallelism, but reduces the number of phases (and hence communication startups). For example, the $X \times Y$ domain can be clustered into independently scheduled rectangles, each of size $m \times n$. Anti-diagonals of these rectangles constitute a phase, and only $X/m + Y/n - 1$ phases are required. m and n should chosen so that anit-diagonals have enough rectangles to distribute among processors.

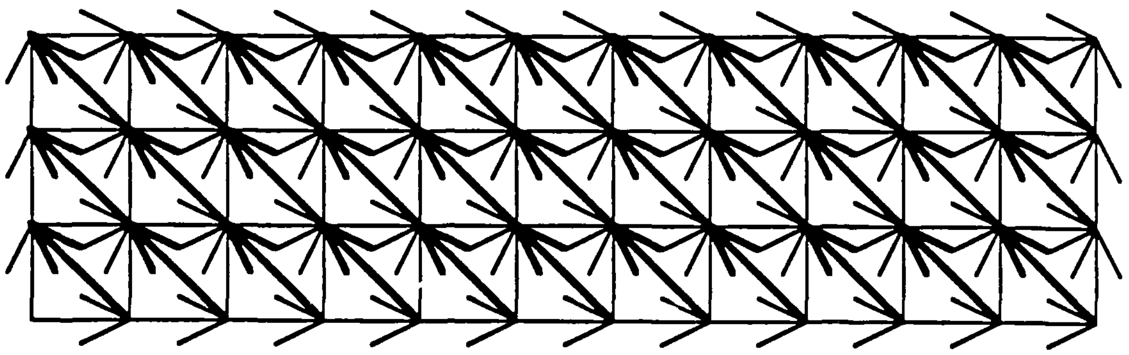


Figure 2: System of Recurrence Equations

Consider a 6 × 6 domain where the solution at a point y depends on the solutions of the two points immediately below, and to the left of y. Figure 5 depicts a clustering that requires only 5 communication startups (as opposed to 11 for the unclustered case). Each aggregate has six variable solutions. Aggregates are labeled with their phase number, where we see that the depicted clustering *completely serializes* the execution. Figure 4 shows an alternate clustering that exploits some parallelism and has two fewer phases.

3 Description of Partitioning Algorithm

We now present a method for partitioning ISRE computations. This method tacitly assumes a two-dimensional problem domain. Generalization of the method to higher dimensions is straightforward. The method presented here can be regarded as an extension of the method presented in [10].

First, an overview. We view the ISRE system as a directed acyclic graph G upon which we apply three distinct operations. First, we partition the nodes into *strings*. Strings are chains of nodes in G, and serve as a generalization of separating hyperplanes. A string is roughly orthogonal to the computational wavefronts. Strings implicitly induce another graph, the *string DAG*; nodes on a common string are represented by one string DAG node. The second operation condenses the string DAG by combining "chains" of string DAG nodes into *tubes*. The third operation applies a clustering algorithm to nodes of the original graph G. Clusters are constrained so that each cluster lies within a single tube.

We will use a running example to illustrate how we partition a problem. Figure 6 depicts a DAG arising from a system of uniform recurrence equations that might have been obtained from a zero fill incomplete factorization of a matrix arising from the discretization of an partial differential equation. Note that the domain of these recurrence equations is somewhat irregular.

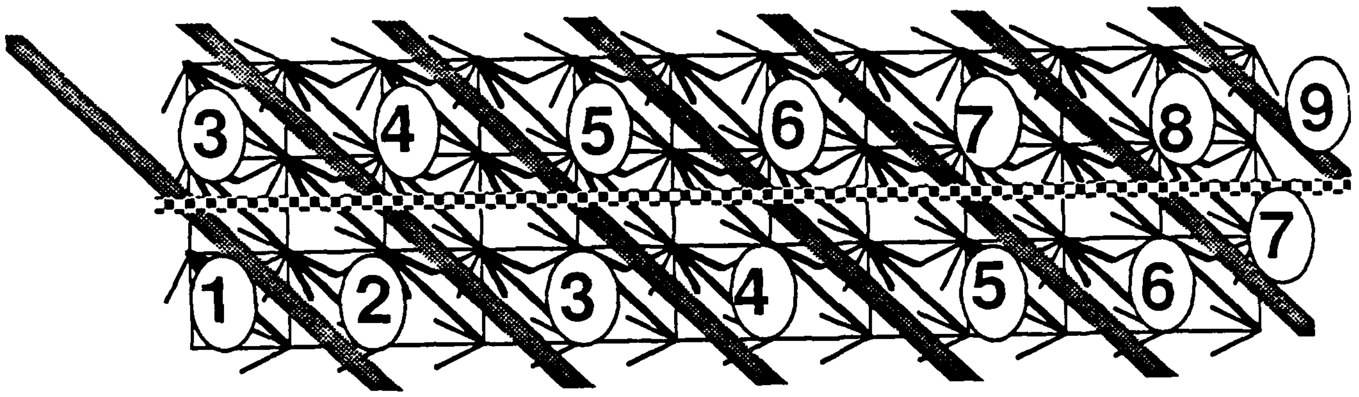


Figure 3: Partitioned Recurrence Equations

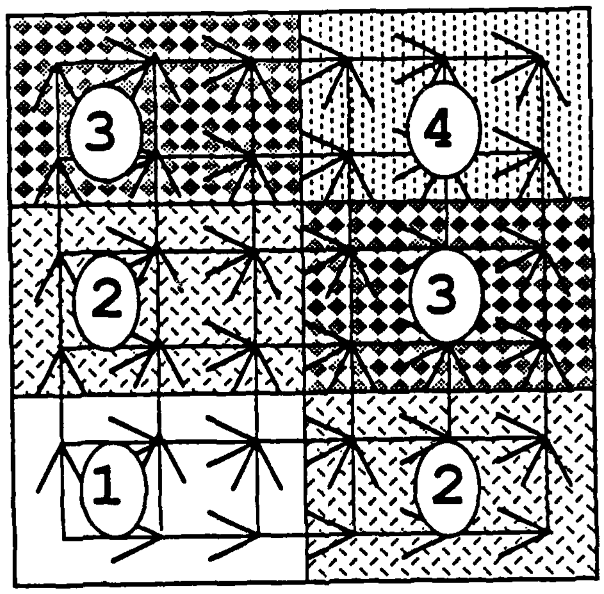


Figure 4: Favorable Partition

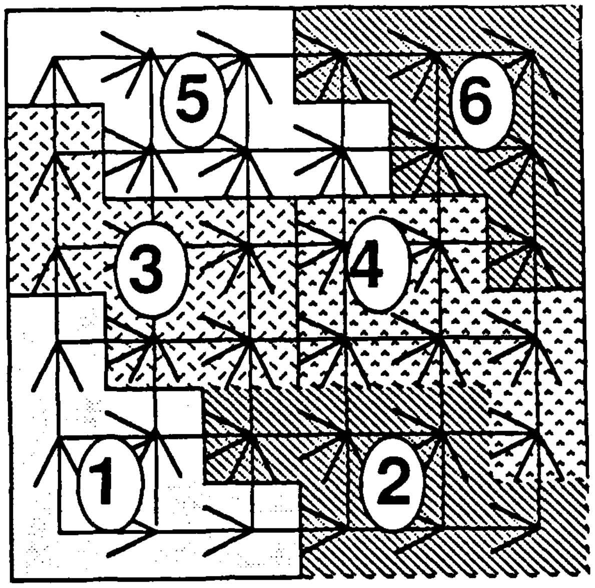


Figure 5: Unfavorable Partition

3.1 Formation of Strings

We will first describe the process of forming strings. The string partition of the example DAG is depicted in Figure 7. A start vertex of G is defined as a vertex not pointed to by any edge. The vertices of S are chosen in the following way. A start vertex V of G is picked, all edges emanating from V are removed; if a new start vertex V' is created through the removal of edges, V' is included in the string. The process is continued recursively to remove as many vertices as possible from G, assigning them to S. When the removal of a vertex exposes multiple start vertices, only one of these start vertices is included in S. As each vertex V' is assigned to S, we mark the vertices W remaining in G that had edges arising from V'. New strings are begun using available start vertices. In picking vertices to incorporate in all strings after the first, priority is given to vertices previously marked by other strings.

Strings have the following properties: (1) The points in each string are connected, (2) There is no more than one point belonging to a given wavefront in a given string, (3) The graph describing the *inter-string* dependencies is a directed acyclic graph and is called the *string DAG*.

In many cases the vertices in the DAG G are ordered in a manner that conveys geometrical meaning. Frequently when matrices are generated from reasonably regular domains, points in a domain are numbered in a very systematic manner. For instance in a rectangular domain, the numbering would begin in a domain corner and a strip of points along one domain edge would be numbered first. Consecutive strips of points in the domain would then be systematically numbered. Typically, for a given problem, strings can be chosen in a number of different ways. In domains that have been numbered in a systematic manner, it can be advantageous to maintain a convention that prescribes how strings should be oriented. We choose strings that follow the pattern of domain ordering, i.e. when more than one start vertex is created though the removal of edges, we choose the start vertex with the lowest number.

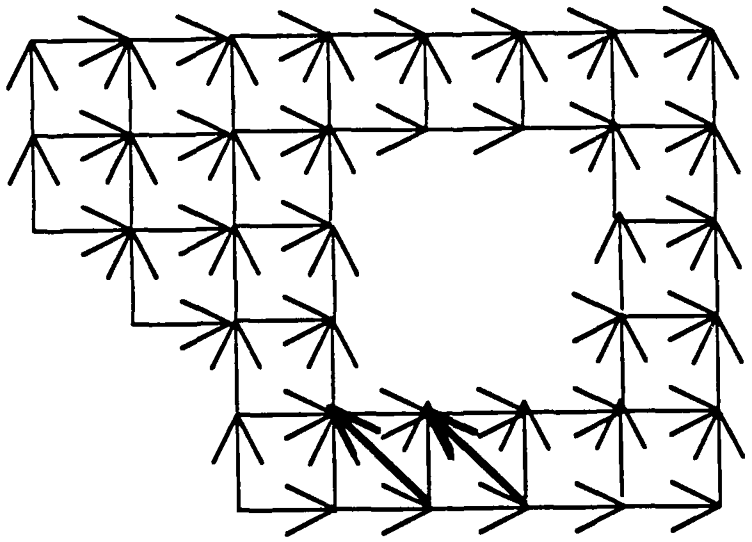


Figure 6: Example DAG

3.2 Formation of Tubes from String DAG

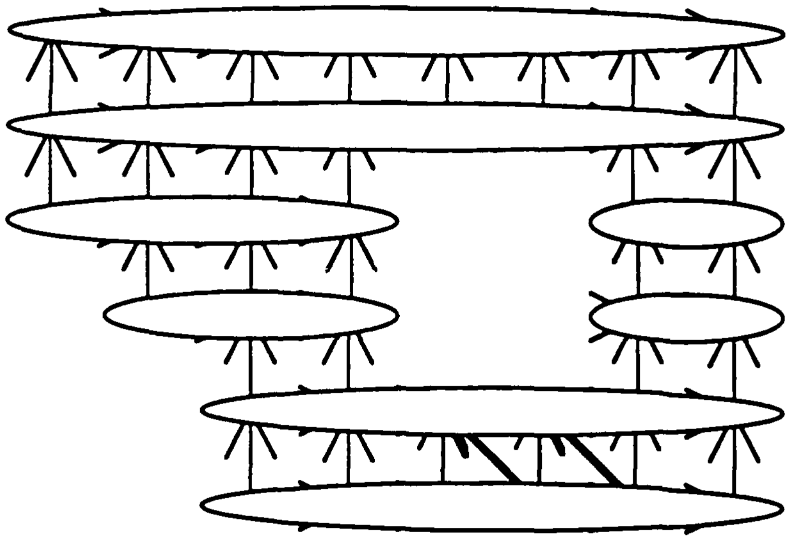
We next perform a depth first traversal of the string DAG; in this traversal a node of the string DAG can be visited only after all of its predecessors have been visited. During this traversal, the string DAG is partitioned into a set of consecutively ordered *tubes.* A maximum of b strings are assigned to each tube; a new tube is begun whenever the depth first traversal backs up and begins a new path. In the running example we depict a string DAG and a partition of that DAG into tubes. (Figure 7).

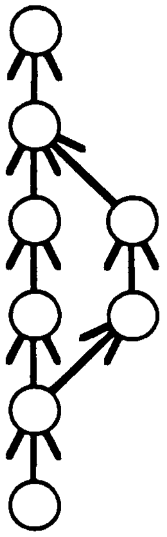
3.3 Formation of Clusters

The tubes identified above will now be used to constrain the choice of nodes that can be used in the formation of clusters. Because each cluster is scheduled in an indivisible manner, we need to be very judicious in how we choose vertices to incorporate into a cluster. Recall how Figure 5 illustrated a choice of clusters that resulted in a completely sequential execution. To reduce the sequentializing effects of inter-cluster dependencies, we add points so that each cluster is constrained to lie within a tube. Figure 4 illustrated a choice of clusters satisfying this constraint. The tubes of clusters produced here are analogous to the *tubes* described in section 2.

We want to choose clusters so that we reduce communication costs while maintaining as much concurrency as we can. To motivate our methods for doing this, note how the clusters are chosen in figure 3. Each tube could be sliced into lines of points forming separating hyperplanes, in this example the new separating hyperplanes are at a 120 degree angle to the strings used in forming the tube. We choose how many strings will be grouped together to form a cluster in the first tube. For the problem in figure 3, the choice of how to group points into clusters in the other tubes follows naturally from the choice made for the first tube.

Vertices that have no incoming edges arising from strings other than their own will be termed *boundary* vertices. Vertices within a tube are incorporated into clusters in a fixed order. During the





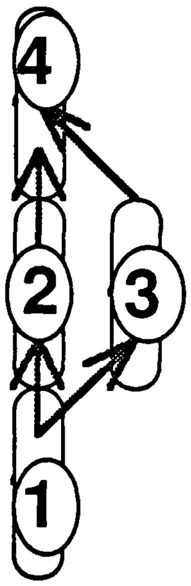


Figure 7: String Partition, String DAG and Tube DAG

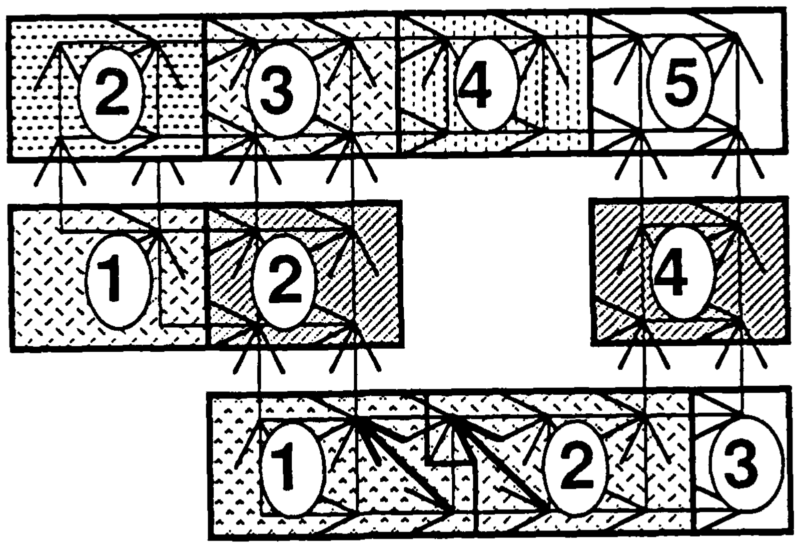


Figure 8: Cluster DAG for Irregular Problem

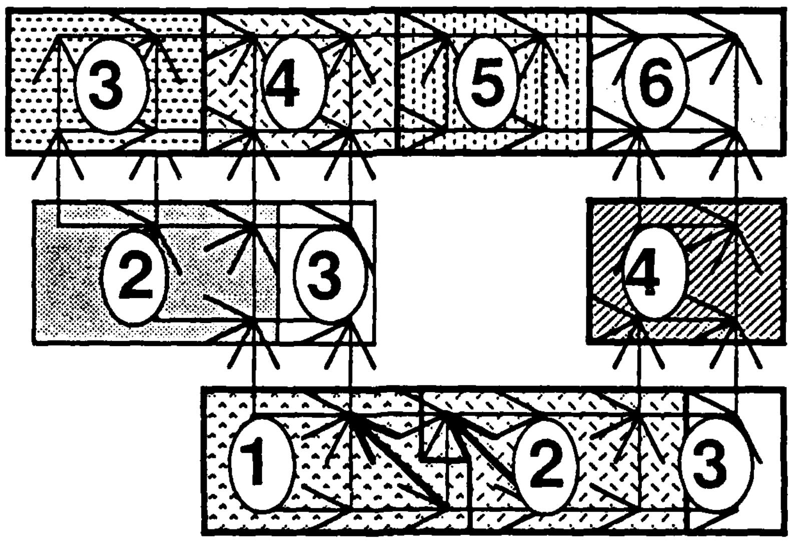


Figure 9: Disadvantageous Cluster DAG for Irregular Problem

normal operation of the clustering process, each new vertex inspected can either be incorporated in the cluster currently being form or can be used to begin a new cluster. The rules for deciding whether to incorporate a vertex into a cluster depend on whether the vertex in question is a boundary vertex. While all vertices in the first string of the string DAG are by definition boundary vertices, vertices in other strings may be boundary vertices as well.

First consider how to go about deciding which non-boundary vertices to place in a cluster. There is a DAG describing the data dependency relations between clusters. We do not want to delay work involving any vertex V in a cluster C because of a possible dependence of another vertex V' in C on some cluster in the cluster DAG. While clusters in 3 and 8 satisfy this property, the clusters in figure 9 do not. We thus plan to incorporate into the cluster all vertices in the given tube that do not have problematic dependencies on other clusters. In uniform recurrence equations in a quadrant, the above constraint corresponds to matching up separating hyperplanes in each tube to obtain a separating hyperplane of the domain as a whole. In regular problems, these constraints lead to clusters of close to uniform size, for the sake of robustness one still specifies a maximal cluster size that cannot be exceeded.

When we create clusters using boundary vertices, there is no natural constraint on cluster size. We still need a mechanism to make sure that the cluster is appropriately shaped, this is described in the detained algorithm description below. The overall granularity of the partition is determined by the sizes of these clusters. This process is apparent in 4, where we choose a size of six for the clusters on the boundary. Clusters obtained from Figure 7 are depicted in Figure 8. The clusters labelled with the number 1 are the boundary clusters.

3.4 Detailed Description of Clustering

A more precise description of the clustering algorithm will now be presented. First some definitions. We define the cluster wavefront for cluster C_i to be the wavefront of C in the cluster DAG.

The *induced wavefront* of a vertex v is obtained as follows. Find the set of vertices U on which v depends. Each of the vertices u_i in U must already be assigned to some cluster C_j. The induced wavefront of v is equal to one plus the maximum of the cluster wavefronts of the clusters $C_j \neq C_i$. If all vertices in a cluster have the same induced wavefront, we obtain a partition like figure 8. In contrast, in figure 9, we note that the cluster wavefront of an entire cluster must be increased due to data dependencies involving just two vertices.

A vertex V is *fair game* for inclusion into a cluster C if 1) V has no unincorporated predecessors, 2) the *induced wavefront* of V is less than or equal to that of C and 3) V belongs to the same tube of strings as any other vertices already belonging to C.

We now present the method used to cluster vertices. Start with the highest numbered string in a tube with a *fair game* vertex. This is the *lead string.* Remove a vertex from the lead string and then remove all *fair game* vertices created. Continue this process until there are no more *fair game* vertices in the lead string. At this point, we seek a new lead string (again chosen as the highest numbered string in the tube having a fair game node) and continue the clustering processes.

The above process is terminated under the following conditions: (1) there are no unincorporated *fair game* vertices (2) the next vertex to be incorporated is a boundary vertex in a lead string and the cluster size is larger than a specified constant K_1. (3) The cluster size is larger than another specified constant K_2.

The constant K_1 is used to specify granularity while the constant $K_2 \geq K_1$ is used as a backup condition to prevent the formation of extremely large clusters in irregular graphs.

In Figure 8, we have used $K_1 = 4$. Note that this does *not* mean that the clusters are all of size 4.

3.5 Multiprocessor Mapping and Experimental Data

The cluster DAG, once formed must be mapped to the fragmented memory multiprocessor. The computational work can be mapped to a multiprocessor in a variety of ways. One simple manner of mapping work is to assign all clusters in consecutively numbered tubes to (where architecturally feasible) physically adjacent processors. If there are more tubes than processors, the tubes are assigned to processors in a wrapped manner. To obtain satisfactory results, it is essential that the program that carries out the work specified by the clustering algorithm be constructed with great care. We have written an extensively tested such a program on the Intel iPSC/2.

We have found that the optimizations described here can make a very substantial impact on performance in solving sparse triangular systems that arise in the solution of a variety of partial differential equations [1] , [9] and [2]. we have documented, for a variety of sparse triangular systems, three fold differences in iPSC/2 execution times.

4 Conclusion

In many modern algorithms, relatively regular problems are encoded using flexible general purpose data structures. To obtain satisfactory performance on distributed memory architectures, it is often necessary to reconstruct and exploit the underlying dependency structure. We have presented and illustrated a method to partition loops that have runtime dependencies that resemble uniform recurrence equations. An extremely important example of implicitly specified recurrence equations are encountered when solving sparse triangular systems arising from incomplete factorizations of matrices formed from meshes of partial differential equations.

References

[1] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA*, January 1988.

[2] K. Crowley, J. Saltz, R. Mirchandaney, and H. Berryman. *Run-time Scheduling and Execution of Loops on Message Passing Machines.* Report 89-7, ICASE, January 1989.

[3] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational iterative methods for non-symmetric systems of linear equations. *Siam Journal on Numerical Analysis*, 1983:345–357, 1983.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[5] L. Hyafil and H.T. Kung. The complexity of parallel evaluation of linear recurrences. *JACM*, 24, 1977.

[6] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, July 1967.

[7] H. T. Kung. In Parter, editor, *Systolic Algorithms*, 1984.

[8] L. Lamport. The parallel execution of do loops. *CACM*, 17, 1974.

[9] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principals of run-time support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, July 1988.

[10] J. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, to appear, 1989.